applcm

applcm主要执行包的实例化工作。在业务流程中，该组件会在“**沙箱部署**”与“**边缘节点（mecHost）部署**”场景下被调用。

applcm位于**mepm侧(注意，eg的mepm位于偏边缘侧，mepm的注册位于边缘节点管理处)**，具体由**k8splugin**和**lcmcontroller**组成，lcmcontroller接收appo或developer的http请求，通过accessToken认证鉴权，解析csar包获取要部署的mecHost，通过adapter使用grpc(作为grpc客户端)调用adapter具体实现（目前为k8splugin）k8splugin作为server接收后实例化。

lcmcontroller

主要用于定义lcm涉及的接口，做认证鉴权和参数解析，涉及的接口包括：

Applcm Interfaces

- Upload Config File
- Delete Config File

实例化app接口

- Instantiate Application
- Terminates Application
- Query
- Query Kpi
- Query MepCapabilities
- Get Mep Capability
- Queries liveness & readiness
- App Deployment Status
- Query workload
- Query AppInstance information
- Query app instances records
- Query stale records

casr包管理

- Upload package
- Delete package

- Delete application package on host
- Distribute package
- Query
- Distribution status
- Sync app package records
- Sync app package stale records
- Add MEC host
- Update MEC host
- Query MEC hosts
- Delete MEC host
- Batch terminate application
- Sync mec host records
- Sync mec host stale records

镜像管理

- Create Image
- Delete Image
- Get Image
- Get Image file
- Get Services
- Get Kong Logs
- Get Subscribe Statistic

下面以实例化app接口为入口，解析其实现，代码位于lcmcontroller/controllers/lcm.go。

```
// @Title Instantiate application
// @Description Instantiate application
// @Param hostIp body string true "hostIp"
// @Param appName body string true "appName"
// @Param packageId body string true "packageId"
// @Param tenantId path string true "tenantId"
// @Param appInstanceId path string true "appInstanceId"
// @Param access_token header string true "access token"
// @Success 200 ok
 // @Failure 400 bad request
 // @router /tenants/:tenantId/app_instances/:appInstanceId/instantiate [post]
func (c *LcmController) Instantiate() {
  //...request ip check
```

```
  ...
  accessToken := c.Ctx.Request.Header.Get(util.AccessToken)
  //...parse request
  var req models.InstantiateRequest
  err = json.Unmarshal(c.Ctx.Input.RequestBody, &req)
  //...err handler
  ...
  bKey := *(*[]byte)(unsafe.Pointer(&accessToken))
  //验证token，得到实例化所需的参数
  appInsId, tenantId, hostIp, packageId, appName, err :=
c.validateToken(accessToken, req, clientIp)
  ...
  originVar, err := util.ValidateName(req.Origin, util.NameRegex)
  ...
  //因为在实例化接口被调用之前，csar包的上传接口已经被提前调用(见developer-be的
沙箱部署),因此DB中已有记录，在此做验证
  appPkgHostRecord := &models.AppPackageHostRecord{
  PkgHostKey: packageId + tenantId + hostIp,
  }
  readErr := c.Db.ReadData(appPkgHostRecord, util.PkgHostKey)
  //...empty & status check，如果包状态不是“Distributed”，抛错
  ...
  appInfoRecord := &models.AppInfoRecord{
  AppInstanceId: appInsId,
  }
  //根据appInstId读取DB中存的appInfo记录，如果已存在，说明被实例化过，抛错
  readErr = c.Db.ReadData(appInfoRecord, util.AppInsId)
  if readErr == nil {
  c.HandleLoggingForError(clientIp, util.BadRequest,
  "App instance info record already exists")
  util.ClearByteArray(bKey)
  return
  }
  //根据hostIp，从DB中获取mecHost的记录，判断其IaaS类型，k8s 还是 vm
  vim, err := c.getVim(clientIp, hostIp)
  ...
  //从环境变量，获取plugin地址addr:port
  pluginInfo := util.GetPluginInfo(vim)
  //创建plugin的grpc客户端
  client, err := pluginAdapter.GetClient(pluginInfo)
```

```
52  //...生成ak sk
53  err, acm := processAkSkConfig(appInsId, appName, &req, clientIp, tenant
Id)
54  ...
55
```

具体看一下AkSk的处理：

```
1  // Process Ak Sk configuration
2  func processAkSkConfig(appInsId, appName string, req *models.InstantiateR
equest, clientIp string,
3   tenantId string) (error, config.AppConfigAdapter) {
4   var applicationConfig config.ApplicationConfig
5   //初始化一个appAuthConfig，如果实例化请求中如果没有携带ak sk，则lcm生成后赋值
6   appAuthConfig := config.NewAppAuthCfg(appInsId)
7   if req.Parameters["ak"] == "" || req.Parameters["sk"] == "" {
8   err := appAuthConfig.GenerateAkSK()
9   ...
10   req.Parameters["ak"] = appAuthConfig.Ak
11   req.Parameters["sk"] = appAuthConfig.Sk
12   req.AkSkLcmGen = true
13   } else {
14   appAuthConfig.Ak = req.Parameters["ak"]
15   appAuthConfig.Sk = req.Parameters["sk"]
16   req.AkSkLcmGen = false
17   }
18   //解析csar包，获取appConfigFile
19   appConfigFile, err := getApplicationConfigFile(tenantId, req.PackageId)
20   ...
21   configYaml, err := os.Open(PackageFolderPath + tenantId + "/" + req.Pac
kageId + "/APPD/" + appConfigFile)
22   ...
23   data, err := yaml.YAMLToJSON(mfFileBytes)
24   ...
25   err = json.Unmarshal(data, &applicationConfig)
26   ...
27   //封装appConfigAdapter,包括了auth信息和基本信息
28   // type AppConfigAdapter struct {
29   //AppAuthCfg AppAuthConfig
30   //AppInfo AppInfo
31   //}
32   acm := config.NewAppConfigMgr(appInsId, appName, appAuthConfig, applica
tionConfig)
```

```
33  //从环境变量读取 APIGW_ADDR，调用apigw的PUT /mep/appMng/v1/applications/{appInstanceId}/confs 配置app的Auth信息
34  err = acm.PostAppAuthConfig(clientIp)
35  ...
36  return nil, acm
37 }
```

继续回到上层，执行db记录和调用plugin执行实例化：

```
1  //更新该租户的记录条目信息
2  err = c.insertOrUpdateTenantRecord(clientIp, tenantId)
3  ...
4  var appInfoParams models.AppInfoRecord
5  appInfoParams.AppInstanceId = appInsId
6  appInfoParams.MecHost = hostIp
7
8  appInfoParams.TenantId = tenantId
9  appInfoParams.AppPackageId = packageId
10  appInfoParams.AppName = appName
11  appInfoParams.Origin = req.Origin
12  //添加DB记录
13  err = c.insertOrUpdateAppInfoRecord(clientIp, appInfoParams)
14  ...
15  //grpc调用plugin，执行实例部署
16  adapter := pluginAdapter.NewPluginAdapter(pluginInfo, client)
17  //内部具体调用 status, err := c.client.Instantiate(ctx, tenantId, accessToken, appInsId, req)
18  err, status := adapter.Instantiate(tenantId, accessToken, appInsId, req)
19  util.ClearByteArray(bKey)
20  ...
21  c.handleLoggingForSuccess(clientIp, "Application instantiated successfully")
22  c.ServeJSON()
23 }
```

k8splugin

k8splugin作为实例化应用的服务端， 提供grpc接口，负责对k8s平台的应用进行部署编排。

接口定义

grpc接口proto定义k8splugin/internal/lcmservice/lcmservice.proto：

```
1 service AppLCM {
```

```
  rpc instantiate (InstantiateRequest) returns (InstantiateResponse) {}
  rpc terminate (TerminateRequest) returns (TerminateResponse) {}
  rpc query (QueryRequest) returns (QueryResponse) {}
  rpc uploadConfig (stream UploadCfgRequest) returns (UploadCfgResponse) {}
  rpc removeConfig (RemoveCfgRequest) returns (RemoveCfgResponse) {}
  rpc workloadEvents (WorkloadEventsRequest) returns (WorkloadEventsResponse) {}
  rpc uploadPackage (stream UploadPackageRequest) returns (UploadPackageResponse) {}
  rpc deletePackage (DeletePackageRequest) returns (DeletePackageResponse) {}
}

service VmImage {
  rpc createVmImage(CreateVmImageRequest) returns (CreateVmImageResponse) {}
  rpc queryVmImage(QueryVmImageRequest) returns (QueryVmImageResponse) {}
  rpc deleteVmImage(DeleteVmImageRequest) returns (DeleteVmImageResponse) {}
  rpc downloadVmImage(DownloadVmImageRequest) returns (stream DownloadVmImageResponse) {}
}
```

服务注册

grpc服务注册位于main.go中的Linsten函数：

```
// Start GRPC server and start listening on the port
func (s *ServerGRPC) Listen() (err error) {
  // Listen announces on the network address
  listener, err = net.Listen("tcp", s.address+":"+s.port)
  ...
  if !s.serverConfig.SslNotEnabled {
  tlsConfig, err := util.GetTLSConfig(s.serverConfig, s.certificate, s.key)
  ...
  // Create the TLS credentials
   creds := credentials.NewTLS(tlsConfig)
   // Create server with TLS credentials
   s.server = grpc.NewServer(grpc.Creds(creds), grpc.InTapHandle(NewRateLimit().Handler))
  } else {
  // Create server without TLS credentials
  s.server = grpc.NewServer(grpc.InTapHandle(NewRateLimit().Handler))
```

```
16  }
17  //将ServerGRPC注册为grpc服务接口的实现
18  lcmservice.RegisterAppLCMServer(s.server, s)
19  // Server start serving
20  err = s.server.Serve(listener)
21  ...
22  return
23 }
```

实例化实现

k8splugin的服务端接口实现位于k8splugin/pkg/server/grpcserver.go，初始化函数为：

```
1  // GRPC server
2  type ServerGRPC struct {
3   server *grpc.Server
4   port string
5   address string
6   certificate string
7   key string
8   db pgdb.Database
9   serverConfig *conf.ServerConfigurations
10  }
11  // Constructor to GRPC server
12  func NewServerGRPC(cfg ServerGRPCConfig) (s ServerGRPC) {
13   s.port = cfg.Port
14   s.address = cfg.Address
15   s.certificate = cfg.ServerConfig.CertFilePath
16   s.key = cfg.ServerConfig.KeyFilePath
17   s.serverConfig = cfg.ServerConfig
18   dbAdapter, err := pgdb.GetDbAdapter(cfg.ServerConfig)
19   ...
20   s.db = dbAdapter
21   return
22  }
```

DB操作

以实例化函数入口实现为例：

```
1  func (s *ServerGRPC) Instantiate(ctx context.Context,
2   req *lcmservice.InstantiateRequest) (resp *lcmservice.InstantiateRespons
e, err error) {
3   //init 返回数据，打印clientIP等信息，解析请求参数
4   ...
```

```
    err = s.displayReceivedMsg(ctx, util.Instantiate)
    ...
    tenantId, packageId, hostIp, appInsId, ak, sk, err := s.validateInputParamsForInstantiate(req)
    ...
    appPkgRecord := &models.AppPackage{
     AppPkgId: packageId + tenantId + hostIp,
     }
     //读取csar包记录
     readErr := s.db.ReadData(appPkgRecord, util.AppPkgId)
     ...
     // 返回一个封装的HelmClient，其定义为：
     // type HelmClient struct {
     // HostIP string
     // Kubeconfig string
     //}
     client, err := adapter.GetClient(util.DeployType, hostIp)
     //执行deploy
     releaseName, namespace, err := client.Deploy(appPkgRecord, appInsId, ak, sk, s.db)
     //...
     err = s.insertOrUpdateAppInsRecord(appInsId, hostIp, releaseName, namespace)
     resp.Status = util.Success
     s.handleLoggingForSuccess(ctx, util.Instantiate, "Application instantiated successfully")
     return resp, nil
}
```

Helm Deploy

进入Deploy函数，其实现即调用helm的sdk去部署chart包：

```
// Install a given helm chart
func (hc *HelmClient) Deploy(appPkgRecord *models.AppPackage, appInsId, ak, sk string, db pgdb.Database) (string, string, error) {
    //解析csar包的helm chart包
    helmChart, err := hc.getHelmChart(appPkgRecord.TenantId, appPkgRecord.HostIp, appPkgRecord.PackageId)
    tarFile, err := os.Open(helmChart)
    ...
    //封装一个appAuthConfig对象，该对象实现了将ak sk信息填写入chart包的values.yaml
    appAuthCfg := config.NewBuildAppAuthConfig(appInsId, ak, sk)
```

```
9   //解析chart包，将sk sk赋值给chart的values.yaml
10  dirName, namespace, err := appAuthCfg.AddValues(tarFile)
11  //...log
12  // load chart包至一个chart结构体
13  chart, err := loader.Load(dirName + ".tar.gz")
14  ...
15  //如果ns不是default，首先使用client-go创建ns
16  if namespace != util.Default {
17  // uses the current context in kubeconfig
18  kubeConfig, err := clientcmd.BuildConfigFromFlags("", hc.Kubeconfig)
19  clientSet, err := kubernetes.NewForConfig(kubeConfig)
20  nsName := &corev1.Namespace{
21  ObjectMeta: metav1.ObjectMeta{
22  Name: namespace,
23  },
24  }
25  ...
26  _, err = clientSet.CoreV1().Namespaces().Create(context.Background(), n
sName, metav1.CreateOptions{})
27  }
28
29  // Release name will be taken from the name in chart's metadata
30  relName := chart.Metadata.Name
31  ...//check db,  在db中检查appName是否已存在，存在说明已被初始化，报错
32  ...
33  // Initialize action config, 调用helm sdk
34  actionConfig := new(action.Configuration)
35  if err := actionConfig.Init(kube.GetConfig(hc.Kubeconfig, "",
namespace), namespace,
36  util.HelmDriver, func(format string, v ...interface{}) {
37  //log func define...
38  });...
39  }
40
41  // Prepare chart install action and install chart
42  installer := action.NewInstall(actionConfig)
43  installer.Namespace = namespace
44  // so if we want to deploy helm charts via k8splugin..
45  //first namespace should be created or exist then we can deploy helm ch
arts in that namespace
46  installer.ReleaseName = relName
```

```
47  //直接调用helm sdk的run
48  rel, err := installer.Run(chart, nil)
49  //... if err, uninstall app
50  //return appName, namespace
51  return rel.Name, namespace, err
52 }
```